LTC: a Fast Algorithm to Accurately Find Significant Items in Data Streams

Shiyu Cheng*†, Dongsheng Yang*†, Tong Yang✉†‡, Haowei Zhang†, and Bin Cui†§¶

Abstract—Finding top-k frequent items has been a hot issue in databases. Finding top-k persistent items is a new issue, and has attracted increasing attention in recent years. In practice, users often want to know which items are significant, *i.e.*, not only frequent but also persistent. No prior art can address both of the above two issues at the same time. Also, for high-speed data streams, prior art cannot achieve high accuracy when the memory is tight. In this paper, we define a new issue, named finding significant items, and propose a novel algorithm namely LTC to address this issue. It includes two key techniques, Long-tail Restoring and CLOCK, as well as three optimizations. In addition, LTC is extended to support finding significant items with thresholds. We theoretically derive the correct rate and error bound, and conduct extensive experiments on three real datasets to test the performance of LTC. Our experimental results show that LTC can achieve 10^5 times higher accuracy in terms of average relative error than other related algorithms. Lastly, LTC is applied to a DDoS detection task and it shows that finding significant items is more powerful than finding frequent items.
Index Terms — algorithms, data structures, succinct data structures, data stream processing, heavy hitter

1 Introduction

1.1 Background and Motivations

Nowadays, the volume of data becomes larger and larger, posing great challenges on fast data mining [1]. On the one hand, the exploding data provides more opportunities for users to better understand the world. On the other hand, it becomes harder and harder for users or administrators to find the information that they care about the most timely.

In many scenarios, users care about the top-k items in a data stream or a dataset, and want to get results immediately. For example, people want to know the most influential tweets under a certain topic. For another example, customers want to know which products are sold the best in a certain category.

In the above examples, it is often acceptable to have a little inaccuracy in the answer because of the following two reasons. First, the dataset often contains noise, which means even if the answers are completely correct based on the dataset, there is still error based on the ground truth. Second, such queries are often intermediate results, serving for comprehensive queries, which are not very sensitive to the error in the intermediate results. For example, when people do feature engineering in machine learning, people do not require all the features to be useful, but people hope that the number of useful features is as many as possible. In other words, as long as the error is small enough, it has little impact on the final results. Therefore, approximate query processing has gained rising attention, and a series of approximate data structures have been proposed and have played important roles in the last several decades, including Bloom filter [2] and its variances [3]–[5], and sketches [6]–[11]. There are two advantages of approximate query processing. First, maintaining an approximate data structure requires less memory and less computing resources. Second, approximate data structures can support fast insertion and query, so they can catch up with the fast speed of data streams. In summary, approximate query processing is an effective method for big data applications.

In terms of finding the top-k items, a traditional metric is frequency [12]–[17], while a recently emerged metric is persistency [18], [19]. Existing works only measure one of those two metrics at a time, but we argue that both frequent and persistent contribute to the significance of items and thus need to be considered together. In this paper, we define significant items as follows.

Definition of Significant Items: *Item* is an object with a unique ID. *Entry* is one appearance of an item on a specific timestamp. *Period* is a range of time. A *data stream* or *dataset* is composed of many periods and has many entries. Each item has two metrics: frequency and persistency. *Frequency* refers to the total number of entries of this item. *Persistency* refers to the number of periods containing at least one entry of this item. The *Significance* s of an item is a combination of frequency f and persistency p weighted by user-defined weights α and β.

$$s = \alpha f + \beta p \tag{1}$$

Use Case 1: DDoS attacks detection. Accurate and timely detection of DDoS attacks is a hot issue [20]. The DDoS attackers send overwhelming packets to victims, which can be detected by finding frequent items. However, frequent items can also be benign users who simply have more workload, so it is challenging to separate DDoS attackers from benign users if only frequency is measured. That motivates us to take persistency into the consideration besides frequency. The experimental study presented in Section 6.15 shows that our approach can find DDoS attackers more accurately.

* *Shiyu Cheng and Dongsheng Yang contributed equally to this study.*
† *Department of Computer Science and Technology, and National Engineering Laboratory for Big Data Analysis Technology and Application, Peking University, China*
‡ *PCL Research Center of Networks and Communications, Pengcheng Laboratory*
§ *National Engineering Laboratory for Big Data Analysis Technology and Application (PKU), China*
¶ *Institute of Computational Social Science, Peking University (Qingdao)*
✉ *Corresponding author: Tong Yang yangtongemail@gmail.com.*
The preliminary version of this paper titled "Finding Significant Items in Data Streams" appears in the proceedings of the 35th IEEE International Conference on Data Engineering (ICDE 2019).

Use Case 2: Website evaluation. Evaluating the websites by popularity need to be accurate and fast, and the rank should be updated in real time. There are two key metrics of popularity: frequency and persistency. Frequency refers to the number of the user's accesses to the website, while persistency indicates whether this website is popular all the time. Both of the two metrics should be considered in ranking the popularity of a website. This also useful for analyzing social networks [21], [22].

Use Case 3: Network congestion control. Network congestion happens every second in data centers [23]–[25]. One effective solution for solving congestion is to change the forwarding path of some flows[1]. As there could be millions of flows in every second, a straightforward solution is to change the forwarding paths of many flows, and then many entries of the forwarding table in the switch will be modified. Some algorithms for organizing the forwarding table do not support fast update. To avoid modifying too many entries of the forwarding table, it is highly desirable to change as few flow paths as possible. A classic method is to only change the forwarding path of large flows. This method does not always work well, because the current large flows could be a burst, and there could be very few packets later. Therefore, changing the forwarding entry of such large flows is in vain. A better choice is to detect the significant flows. They are not only frequent, but also persistent, and thus with high probability they will also be large flows in a long period later. Therefore, changing the significant flows has the highest benefits.

1.2 Limitations of Prior Art

Existing solutions focus on only one metric: either finding top-k frequent items [12]–[17], or finding top-k persistent items [18], [19]. However, in practice, finding items that are both frequent and persistent is often essential. One straightforward solution to handle that is to combine the above two kinds of algorithms. Specifically, users build two data structures: one for recording the frequent items, the other for recording the persistent items. Obviously, the time and space overhead is the sum of the overhead of two algorithms. Also, suppose there is an item that is frequent but not persistent, which thus should not be recorded. Since the two data structures are separated, the frequency data structure does not know it is not persistent and will record this item, which is a waste of memory. Therefore, existing solutions are inefficient for finding significant items.

1.3 Our Solution

The goal of this paper is to propose a solution to accurately and quickly find significant items with limited memory. To achieve that, we propose a novel algorithm, namely Long-Tail Clock (LTC), which contains two key techniques: Long-tail Restoring and an adapted CLOCK algorithm.

First, we show how *Long-tail Restoring* works. To make it easier to understand, we use the frequency as the metric. A common setting for this problem is use an array of k cells to keep track of frequent items. Each cell stores an item ID and its frequency $\langle e, f \rangle$. The key issue is how to handle a new item when the k cells are full. The most well known algorithm, Space-Saving, simply makes the incoming item replace the smallest item $\langle e_{min}, f_{min} \rangle$ among all k items in those cells, and sets the initial value of the new incoming item to $f_{min} + 1$. It will cause large overestimation error because the replacement happens too easily. In contrast, our LTC works as follows. When a new item arrives, it counter-weights f_{min} by 1. And when f_{min} is deducted to 0, e_{min} is replaced by the new item. It is important to restore the values of the new item that is used to counterweight the old item to mitigate underestimation. Our algorithm is based on the observation that item frequencies in real datasets follow the *long-tail distribution* [26]–[30], which means f_{min} before being deducted is similar to the f of the second smallest item. Therefore, the initial value of the new item is set to the second smallest f minus 1. More details about Long-tail Restoring are presented in Section 4.4.

Second, we proposed an adapted CLOCK algorithm to record persistency. The main issue of maintaining persistency is to increment the persistency exactly *by one* for any item that appears more than once in one period. Our idea is to leverage the spirit of the CLOCK algorithm [31]–[33]. It uses a pointer to mark periods and does not need any additional information, so it saves memory. There are two versions of our CLOCK algorithms: a basic version (Section 4.2) with overestimation error and an optimized version (Section 4.3) eliminating the overestimation error.

In addition, we propose optimizations including extension of significance (Section 4.5), frequency based replacement (Section 4.6), and two-level structure (Section 4.7), which can make LTC perform better on specific tasks. Moreover, we address the problem called finding significant items with thresholds: frequency is at least x and persistency is at least y. This is different from finding top-k significant items because the number of target items is unknown in advance. Our LTC works well for this application with an additional array as is discussed in Section 5.

2 Related Work

To the best of our knowledge, there is no prior work for finding top-k significant items. A straightforward method is to combine two kinds of algorithms: one for finding frequent items and the other for finding persistent items. Here we briefly describe them.

2.1 Finding Top-k Frequent Items

For finding top-k frequent items, existing algorithms can be divided into two categories: counter-based and sketch-based.

Counter-based: Counter-based algorithms include Space-Saving (SS) [15], Lossy Counting (LC) [17], CSS [16], etc. These algorithms are similar to each other. Take Space-Saving as an example, it maintains several cells, and each cell records a pair $\langle e_i, f_i \rangle$, where e_i is an item ID and f_i is the estimated frequency of e_i. When an item arrives, it first judges whether this item matches one of the cells. If it matches the j^{th} cell, SS increments f_j by 1. Otherwise, it finds the item whose estimated frequency is the smallest, denoted by $\langle e_{min}, f_{min} \rangle$. Then it replaces e_{min} by the new item ID, and increments f_{min} by 1. It uses a structure named Stream-Summary to accelerate the speed for the above operations.

1. A flow is defined by the five tuples: source IP address, destination IP address, source port, destination port, and protocol.

Sketch-based: Sketch-based algorithms include the Count sketch [6], the CM sketch (CM) [7], etc. They are similar to each other. Take CM as an example, it uses multiple equal-sized columns associated with different hash functions h_i. Each column is comprised of several cells. When an item e arrives, each column first computes $h_i(e)$ to map e to the cell $A[i][h_i(e)]$, and then increments the value of that cell by 1. When e is queried, the estimated frequency is the smallest value of the mapped cells of all the columns. Estan and Varghese proposed the CU sketch (CU) [8]. It improves CM by incrementing only the minimum value(s) among the mapped cells by 1 instead of incrementing all the values of mapped cells when an item arrives. To report the top-k frequent items, Sketch-based algorithms need to maintain a min-heap to record and update top-k frequent items.

2.2 Finding Top-k Persistent Items

For finding top-k persistent items, there are several existing algorithms, such as coordinated 1-sampling [19], PIE [18] and its variant [34]. Because coordinated 1-sampling focuses on improving the performance of distributed data streams, we do not introduce it in detail. The state-of-the-art algorithm is PIE. The key idea of PIE is to use Raptor codes [35] to record and identify item IDs. It executes each period one by one. During each period, it maintains a data structure called Space-Time Bloom Filter and uses Raptor codes to encode the IDs of items appeared in this period. Finally, it gathers all Space-Time Bloom Filters and decodes each item by the recorded raptor codes.

Besides, we can adapt sketch-based algorithms to finding top-k persistent items. The thorniest problem is that some items might appear more than once in one period, which means we cannot update the persistency directly. To deal with this problem, we maintain a standard Bloom filter (BF) [2], [36] to record whether it has appeared in the current period. We also need to maintain a min-heap to assist in finding top-k persistent items.

3 Overview of LTC

In this section, we outline the general concept of our proposed algorithm, Long-tail Clock Sketch (LTC). There are multiple LTC implementations described in Section 4, but they all follow the framework described in this section.

3.1 Data structure

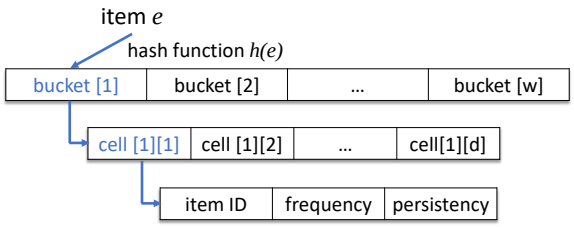


Fig. 1: The data structure of LTC.

As shown in Figure 1, the data structure of LTC is a hash table comprised of w buckets. There is a hash function h(.) to map each item to a bucket. Each bucket is comprised of d cells. Each cell is either empty or occupied by a triple: $\langle ID, frequency, persistency \rangle$. The ID field stores the ID of the item in this cell; The frequency field stores the estimated total number of appearances of this item in all time periods; The persistency field stores the estimated number of the time periods where this item appears, and it also maintains the necessary information to mark a period.

3.2 Operations

3.2.1 Initialization

A cell is called empty if its ID field is empty and both frequency and persistency equal to 0. We initialize LTC by setting all the cells in all the buckets to empty.

3.2.2 Insertion

For convenience, we use $A[i][j]$ to denote the j^{th} cell of the i^{th} bucket in the hash table. For each incoming item e, LTC first computes a hash value $h(e)$ with the hash function $h(.)$. Then it maps e to bucket $A[h(e)]$ and checks all the cells in $A[h(e)]$. There are three cases as follows:

Case 1: e is found in a cell of $A[h(e)]$, denoted by $A[h(e)][j]$. In this case, LTC inserts e into $A[h(e)][j]$ with the **cell update** operation.

Case 2: e is not found in any cell of $A[h(e)]$, but there is an empty cell $A[h(e)][j]$. In this case, LTC inserts e into $A[h(e)][j]$ with the **cell update** operation.

Case 3: e is not found in $A[h(e)]$, and there is no empty cell in $A[h(e)]$. In this case, LTC first finds the smallest cell in $A[h(e)]$. And then it performs the significance decay operation on this cell, which is to decrease both of the frequency and persistency by 1. After that, if the significance is decreased to 0, LTC replaces the item in the smallest cell with e and sets the initial frequency and persistency to 1.

3.2.3 Query

To query for a specific item e, LTC first computes the bucket index $h(e)$ with the hash function, and then checks all cells in bucket $A[h(e)]$ to see if any cell's ID field matches with e. If there is a match, the frequency, persistency and significance of e are returned. If there is no match, e is reported as an insignificant item.

3.3 Properties of LTC Framework

3.3.1 High Memory Efficiency

The idea of LTC is to only maintain the significant items and the items with high potential to be significant. To be specific, all items in a bucket except the one in the smallest cell have a large probability to be the most significant items among all the items mapped to this bucket. The smallest cell, on the other hand, holds the item with high potential to be significant. With such a design, LTC discards most of the insignificant items and saves much memory.

3.3.2 High Scalability

For prior art, such as Lossy Counting and Space-Saving, the overhead of insertion grows as the data structure size grows, because they need to match items across the whole data structure. By contrast, LTC matches items only inside a bucket. For LTC, the overhead of insertion is composed of two parts, finding a bucket and finding a cell. Finding a bucket is conducted by calculating a hash function, which is an O(1) operation. Finding a cell is done by matching with each cell in the bucket, which is an O(d) operation. Therefore, the total time overhead of inserting an item is O(d). When there are more items, LTC fixes d and increases w, which will not slow down the insertion speed.

4 Implementation of LTC

In this section, we will present the detail of **cell structure** and **cell update**. We start from a naive version, and then introduce the optimizations.

4.1 Naive Version

4.1.1 Cell Structure

The cell in Figure 1 is materialized as follows. The frequency field contains a frequency counter. The persistency field contains a persistency counter and a timestamp. The timestamp stores the most recent time period when the item appears.

4.1.2 Cell Update

When an incoming item is inserted into a cell, LTC performs cell update operation as follows: first, the frequency counter of the cell is incremented by 1. Then, LTC checks the timestamp in the persistency field. If the timestamp is the current time period, it means the item has appeared in this time period, so the persistency counter will not be incremented. Otherwise, LTC increments the persistency counter by 1 and set the timestamp field to the current period.

4.2 Maintaining Persistency using CLOCK

In the naive version, the persistency field uses a timestamp to mark time periods. In this version, we propose to use a Boolean flag to record whether the item has appeared in the current time period, and leverage the well-known CLOCK algorithm [31]–[33] to update the persistency counter periodically. This optimization can save the memory used by the timestamps.

4.2.1 Background

CLOCK [37] is a cache replacement algorithm. It keeps a circular array of the pages. There is a pointer p that scans through this array. When a page in the array is accessed, the flag of the page is set to True. When selecting the page to replace, CLOCK examines the page currently pointed by the pointer: if the flag is True, it resets the flag to False and goes down to the next page; if it finds a page with a False flag, this page is replaced. In LTC, CLOCK is not used as a replacement algorithm. Instead, it represents a period and helps to maintain the persistency.

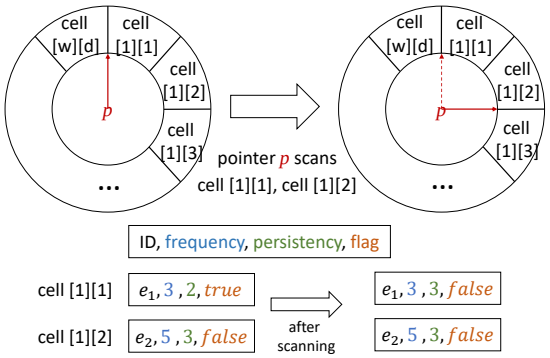


Fig. 2: The CLOCK algorithm in LTC.

4.2.2 Cell Structure

As shown in Figure 2, there are two differences compared with the naive version: (1) All $w \times d$ cells in the table are organized as a CLOCK. There is a pointer p acting as the watch hand of the CLOCK. It moves clockwise and scans the cells one by one; (2) the timestamp in the persistency field is replaced by a Boolean flag.

4.2.3 Cell Update

On the arrival of each item, LTC finds its cell. The cell's frequency counter is incremented by 1 and the flag in the persistency field is set to 1 if it is 0. The cell's persistency counter is not modified. Apart from that, a scan with p is triggered.

Scan: p scans the cells clockwise with a calculated step size, and update the persistency field as follows: if the flag of this cell is false, the persistency field is not modified; if the flag of this cell is true, the persistency counter is incremented by 1, and the flag is reset to false.

Step size of p: The step size of p is carefully set so that p can go from the first cell to exactly the last cell in one time period. Suppose each period lasts for t seconds. LTC maintains the arriving time of the last item, denoted as the y^{th} second. If the current item arrives at the x^{th} second, the step size of the current scan is set to $(x - y)/t * wd$. Note that when the arriving speed of items differs, the step size is adjusted dynamically.

Example: As shown in Figure 2, LTC moves p by 2 steps during a scan. p passes the first two cells in the first bucket, cell [1][1] and cell [1][2], so LTC checks these two cells. For cell [1][1], the flag is true, so its persistency is incremented from 2 to 3, and the flag is reset to false. For cell [1][2], the flag is false, so nothing is changed.

4.3 Eliminating Deviations

The CLOCK optimization saves a lot of memory but causes deviations of periods which can result in inaccurate estimation of persistency. Figure 3 shows an example.

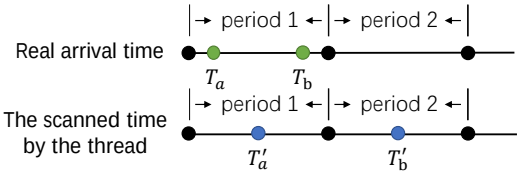


Fig. 3: An example of a deviation.

An item in the middle of the CLOCK is scheduled to be scanned at time T'_a and T'_b (blue points). In this example, it is inserted at T_a and T_b (green points). At time T_a, the flag of the cell is set to true, so the persistency is incremented by 1 at time T'_a. Similarly, at time T_b, the flag is set to true, so the persistency is incremented by 1 at time T'_b. In total, the persistency counter is incremented by 2, but it should be incremented by only 1, because the item only appears in period 1. This error is caused by the deviation between the real period (black points) and the scan period (blue points). Since the scan period is across two real periods, we need to maintain two persistency flags for these two real periods respectively to avoid deviations.

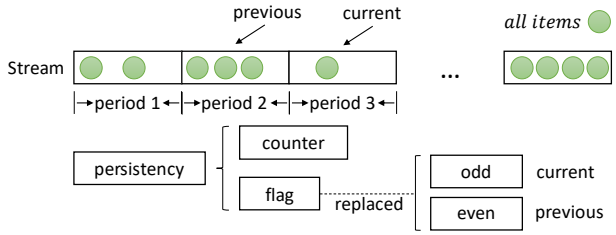


Fig. 4: Eliminating deviations by using two flags.

4.3.1 Cell Structure

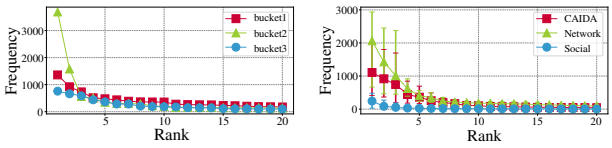
The new cell structure is shown in Figure 4, The flag in the persistency field is now replaced by two flags: an *odd flag* and an *even flag*. The odd flag is for the periods with odd serial number, while the even flag is for the periods with even serial number. Suppose the current period is period 3, then the odd flag in each cell indicates whether the recorded item has appeared in the current period (period 3), and the even flag in each cell indicates whether the recorded item has appeared in the previous period (period 2).

4.3.2 Cell Update

When inserting an item into a cell, the flag which corresponds to the current period is set to 1, and the other flag is not influenced.

During scanning, LTC checks the previous flags of the scanned cells. The previous flag is defined as the odd flag if the current period is an even period, and is defined as the even flag if the current period is a odd period. For each scanned cell, if the previous flag is true, the persistency counter is incremented by 1 and the previous flag is reset to false; otherwise, nothing is modified.

4.4 Long-tail Restoring



(a) Three arbitrary buckets. (b) Three datasets.

Fig. 5: Frequency distribution.

4.4.1 Background

In practice, real datasets usually follow the long-tail distribution [26]–[30]. Furthermore, we show that for LTC, the frequencies of the items mapped to the same bucket also follow the long-tail distribution.

Figure 5 shows experiments on three datasets (detailed in Section 6.2) to visualize the above statement. Figure 5(a) plots the frequencies of the top-20 frequent items in three random buckets for the Net (Network) dataset, and Figure 5(b) plots the average frequencies of the top-20 frequent items in the buckets for each of the three datasets. From both figures, we can see that the low ranking items have similar frequencies, which means that the frequencies of items follow the long-tail distribution.

4.4.2 Long-tail Restoring

Based on the above pattern of datasets, we propose a novel optimization, namely Long-tail Restoring, to substitute for the replacement algorithm in the naive version. For convenience, the values of both persistency counter and frequency counter are collectively called the *value* in this section.

Long-tail Restoring is to set the initial *value* of the newly inserted item to the *value* of the second smallest cell minus 1. The intuition is that a newly inserted item usually comes in a burst, and before it can be inserted into the smallest cell, it countervails the previous item in the smallest cell. In that case, the *value* of the newly inserted item is underestimated by the original *value* of the smallest cell. As mentioned above, real datasets usually follow the long-tail distribution. Therefore, the *value* in the second smallest cell is a good estimation of the original *value* in the smallest cell. By setting the new *value* directly to the second smallest *value* minus 1, the inserted cell is still the smallest cell, and we restore the *value* of the new item as accurately as possible.

The risk of this optimization is low. In the worst case, if an infrequent or inpersistent item is inserted into the smallest cell by accident, the restored *value* could be overestimated. However, with high probability, it will be expelled soon, since it will be deducted when new items come later.

4.5 Optimization I: Extension of Significance

In the basic definition, if an item appears in a period, its persistency will be incremented by 1. However, sometimes users may want an item to accumulate its persistency only if it comes for multiple times in a period. Therefore, we extend the definition of persistency as follows.

Persistency: The persistency of an item is the number of time periods in which the item comes for more than t times.

In addition, the significance s is extended to involve an item-wise weight v. A VIP item can have a higher weight than other items.

Significance: a function of frequency f, frequency weight α, persistency p, persistency weight β, and item weight v.

$$s = v \cdot (\alpha f + \beta p) \tag{2}$$

4.5.1 Cell Structure

The cell structure is similar to the one in Section 4.3, except that the odd and even flag are integers instead of Booleans.

4.5.2 Cell Update

Persistency flags: In even-numbered periods, the even flag of the inserted cell is incremented by one if its value is less than t. In odd-numbered periods, the odd flag of the inserted cell is incremented by one if its value is less than t. The reason we restrict the value to t is to avoid overflow.

Frequency: incremented by v on each insertion.

Scan: In even-numbered periods, LTC checks the odd flags of the scanned cells: for each cell, if the odd flag equals to t, the persistency counter is incremented by v and the odd flag is reset to 0; otherwise, nothing is modified. In odd-numbered periods, LTC handles the even flag similarly.

4.6 Optimization II: Frequency Based Replacement

The main source of under-estimation error of LTC is that the item in the smallest cell of a bucket is deducted when a new item is mapped to this cell. We propose the frequency based replacement method to mitigate this error. The idea of this optimization is to deduct only the frequency instead of both frequency and persistency, and replace the item in the smallest cell when its frequency is deducted to 0. Since the persistency is never deducted, it is much more accurate than the one in the original LTC. Therefore, this optimization benefits applications that emphasize persistency.

The frequency based replacement will have no effect on the replacement, because the persistency will always be smaller or equal to the frequency in an original LTC. First of all, when considering insertion, the persistency of an item will always be equal or smaller than its frequency. Then, considering the deduction, the frequency will be deducted to 0 always after the persistency is already 0 in an original LTC. As a result, an item in an optimized LTC will be replaced at the same time as in an original LTC.

4.7 Optimization III: Two-level Structure

We propose a two-level structure LTC to further mitigate the under-estimation error caused by deduction on the smallest item. The idea is to make the distribution of significant items more balanced in each bucket by moving items from high contention buckets to other buckets.

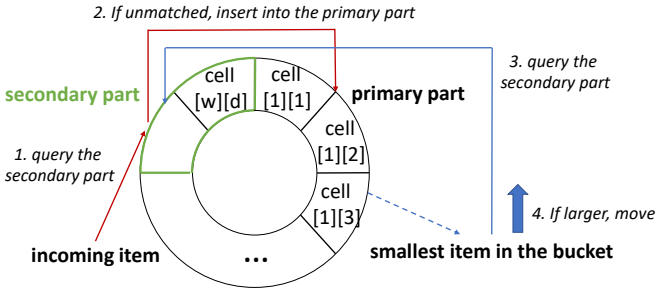


Fig. 6: Structure and pipeline of Two-level LTC.

4.7.1 Data Structure

As shown in Figure 6, the LTC hash table is divided into a primary part and a secondary part. Items in the primary part can be moved into the secondary part, but items in the secondary part cannot be move back to the primary part. The ratio of the size of the two parts is a configurable hyperparameter. These two parts contain the same kinds of buckets and cells. Also, they share the same CLOCK, so the persistency update can be done as usual.

4.7.2 Insertion Pipeline

As shown in Figure 6, there are four potential steps to insert an item: (1) When an item comes, it is first queried in the secondary part by mapping it to its bucket in the secondary part using the hash function. If it exists in a cell of that bucket, it is inserted into that cell and the insertion completes. (2) If not, the next step is to find its bucket in the primary part and insert it. When this item does not exist in the bucket, the smallest item $item_s$ in that bucket is identified. (3) Instead of deducting it directly, LTC maps $item_s$ to its bucket in the secondary part. (4) If an item in that bucket has smaller significance than $item_s$, $item_s$ will replace that item in the secondary part, and LTC will remove $item_s$ from the primary part. Otherwise, $item_s$ is still in the primary part and is deducted as usual.

This optimization can improve the accuracy because an item is exempted from being deducted if it can enter the secondary part, which means LTC now has some extent of global awareness and can rearrange items. However, since there are two parts to check, the throughput of two-level LTC is 50% of the one-level LTC. Therefore, this optimization is recommended only if the throughput is not as important as the accuracy.

4.8 Summary

The standard version of LTC is the naive implementation plus deviation eliminated CLOCK and Long-tail Restoring. There are three additional optimizations that are not implemented in LTC by default: the extension of significance is useful for specific use cases; the frequency based replacement is particularly beneficial when persistency needs to be accurate; the two-level structure is recommended when accuracy is more favored than throughput.

4.9 Mathematical Proofs

In this section, we first derive a theoretical lower bound of the probability that the estimated significance of an item is exactly correct, and then derive the probability that the error of the estimated significance of an item is below a threshold.

4.9.1 The Correct Rate Bound

The correct rate refers to the probability that the estimated significance is exactly equal to the real significance.

We use the following notation: There is a stream $\mathcal{S}$ which has N entries of totally M distinct items: $e_1, e_2, \cdots, e_M$. Let s_i be the significance of e_i in the stream. d is the number of cells in a bucket. w is the number of buckets.

Lemma 4.1. *The estimated significance of an item e is correct if at any time point there are less than $d-1$ items in the same bucket of e and have larger significance than e.*

Proof. When e arrives for the first time, there will be less than $d-1$ items in the bucket mapped by e, so e will find an empty cell in this bucket. Since there are always less than $d-1$ cells in that bucket, e will never be the smallest item in that bucket, so the estimated significance of e will not be deducted by other items. Therefore, the estimated significance of e is always correct. □

We call an item *dangerous* to e if it is mapped to the same bucket as e and its significance has ever been larger than that of e. Let $\wp(e)_i$ denote the probability that e_i is dangerous to e. If $s_i > s$, $\wp(e)_i = \frac{1}{w}$, otherwise $\wp(e)_i = \frac{1}{w} * \frac{s_i}{s+1}$.

Let $p(e)_{j,x}$ denote the probability that for the first j significant items, there are exactly x dangerous items to e. It is a dynamic programming problem [38]–[40] which can be written as follows.

Lemma 4.2.

$$p(e)_{j,x} = p(e)_{j-1,x} * (1 - \wp(e)_j) + p(e)_{j-1,x-1} * \wp(e)_j \quad (3)$$

Proof. If the j^{th} item is dangerous to e, then in order to have exactly x dangerous items in the first j items, the first $j-1$ items should have exactly $x-1$ dangerous items. If the j^{th} item is not dangerous to e, then in order to have exactly x dangerous items in the first j items, the first $j-1$ items should have exactly x dangerous items. □

Now that we have Lemma 4.1 and Lemma 4.2, we can calculate $\mathcal{C}(e)$, which is the correct rate of e.

Theorem 4.1.

$$\mathcal{C}(e) \geqslant \sum_{x=0}^{d-2} p(e)_{M,x} \quad (4)$$

4.9.2 The Error Bound

For an item e_i, let s_i and $\hat{s}_i$ be the real significance and estimated significance, respectively. Assume the items are labelled in the order that $s_1 \geqslant s_2 \geqslant \cdots \geqslant s_M$. Other symbols used below follow the definitions in Section 4.9.1.

For an arbitrary item e_i recorded in LTC, let X_i be the number of times that another item comes to the same bucket as e_i and counterweighs e_i, we have

$$s_i - \hat{s}_i = X_i * (\alpha + \beta) \quad (5)$$

e_i will be counterweighed if and only if its cell is the smallest cell in the bucket. Let P_{small} be the probability that e is the smallest cell. It is equal to having arbitrary $d-1$ items among the $i-1$ items that are larger than e_i, and these $d-1$ items are mapped to the same bucket as e_i, while the rest $i-d$ items are mapped to other buckets.

$$P_{small} = \binom{i-1}{d-1} \left(\frac{1}{w}\right)^{d-1} \left(\frac{w-1}{w}\right)^{i-d} \tag{6}$$

Let V be the number of times e_i is deducted when it is in the smallest cell. The expectation of V is equal to the number of entries mapped to the same bucket as e_i and are less significant than e_i.

$$\mathbb{E}(V) = \frac{1}{w} \sum_{j=i+1}^{M} s_j \tag{7}$$

Then we can derive the formula for $\mathbb{E}(X_i)$.

$$\mathbb{E}(X_i) = P_{small} * \mathbb{E}(V) : \tag{8}$$

According to Equation 5, we have

$$\mathbb{E}(s_i - \hat{s_i}) = P_{small} * \mathbb{E}(V) * (\alpha + \beta) \tag{9}$$

Given an arbitrary positive number ϵ, based on the Markov inequality, the probability that the error of the significance of e_i is larger than e is bounded as follows.

Theorem 4.2.

$$\begin{aligned} Pr\{s_i - \hat{s_i} \geqslant \epsilon\} &\leqslant \frac{\mathbb{E}(s_i - \hat{s_i})}{\epsilon} \\ &\leqslant \frac{P_{small} * \mathbb{E}(V) * (\alpha + \beta)}{\epsilon} \end{aligned} \tag{10}$$

5 Finding Items with Thresholds

In this section, we handle the task of finding significant items with thresholds: finding all items that have frequency at least x and persistency at least y.

5.1 Parameter Mapping

The parameter mapping works as follows. First, the greatest common divisor of x and y is calculated, denoted as g. Second, α is set to x/g, and β is set to y/g. A LTC is built with the above parameters. After inserting all the items into LTC, we examine every item in LTC using the significance threshold, and report the items above the threshold.

The risk of parameter mapping is that LTC sorts items by significance, so the items with the highest significance stay in LTC. However, we want the items with the highest probability to meet the threshold to be in LTC. Despite having the above risk, according to experiments in Section 6.13, it works well in practice.

5.2 Optimization: Auxiliary Array

LTC is not flexible enough because the number of cells is fixed once it is built. When the problem is finding top k items, this is not an issue because we can allocate the size of LTC according to k. However, when the problem is finding all items above the thresholds, it is usually infeasible to estimate the number of significant items. To address this issue, we propose to use an auxiliary array with LTC. When LTC finds an item meeting the thresholds, it moves this item from LTC to the auxiliary array.

5.2.1 Data Structure

In addition to LTC, an auxiliary array is maintained to record the ID of the items meeting the threshold requirement. Also, a Bloom Filter, which performs insertion and query in $O(1)$ time, is associated with the auxiliary array to keep track of the item IDs in the array.

5.2.2 Insertion

Before inserting an item into LTC, this item is queried in the Bloom filter. If the Bloom filter shows that the item is already in the auxiliary array, this item will not be inserted into LTC. Otherwise, it is inserted into LTC. After inserting an item into LTC, its current frequency and persistency are queried and compared to the threshold. If it meets the thresholds, this item is inserted into the auxiliary array and the Bloom filter, and is deleted from LTC.

5.2.3 Query

For queries about the significant items meeting the thresholds, LTC reports all items recorded in the auxiliary array.

5.3 Mathematical Analysis of Recall Rate

The recall rate of this task is essentially the probability of reporting an item which meets the threshold to be significant. We derive a lower bound of the recall rate as follows.

Assume e_i is an item whose real significance rank is i. If e_i meets the threshold requirement, there are two situations that it will be reported to be significant: 1) the cell used to record e_i is not the smallest cell, and 2) the cell used to record e_i is the smallest cell but the number of Significance Decay operations performed on the smallest cell is not enough to make e_i insignificant.

For the first situation, we can derive the probability similar to P_{small} from Equation 6:

$$\sum_{x=0}^{d-2} \binom{i-1}{x} * \left(\frac{1}{w}\right)^{x} * \left(\frac{w-1}{w}\right)^{i-1-x} \tag{11}$$

For the second situation, let V be the number of entries that is mapped to this bucket and counterweighs e_i, we have the same $\mathbb{E}(V)$ with Equation 7.

Every such entry will cause the frequency and the persistency of e_i be underestimated by 1. For convenience, we only analyze frequency here because the analysis of persistency is similar. As long as V is no larger than $f_i - h$, e_i will still be reported as an significant item, where f_i is its real frequency and h is the frequency threshold predefined by users.

According to Markov inequality, we have

$$Pr\{V \geqslant f_i - h + 1\} \leqslant \frac{\mathbb{E}(V)}{f_i - h + 1} \tag{12}$$

Thus, we can derive that

$$\begin{aligned} Pr\{V \leqslant f_i - x\} &\geqslant 1 - \frac{\mathbb{E}(V)}{f_i - h + 1} \\ &\geqslant 1 - \frac{\frac{1}{w}\sum_{j=i+1}^{M} f_j}{f_i - h + 1} \end{aligned} \tag{13}$$

Therefore, the probability that e_i satisfies the second situation is larger than

$$P_{small} * \left(1 - \frac{\frac{1}{w}\sum_{j=i+1}^{M} f_j}{f_i - h + 1}\right) \tag{14}$$

According to the principle of addition, the probability of reporting e_i as a significant item is larger than

$$\sum_{x=0}^{d-2} \binom{i-1}{x} * \left(\frac{1}{w}\right)^{x} * \left(\frac{w-1}{w}\right)^{i-1-x} + P_{small} * \left(1 - \frac{\frac{1}{w}\sum_{j=i+1}^{M} f_j}{f_i - h + 1}\right) \quad (15)$$

6 Experimental Results

In this section, we conduct experiments to show the performance of LTC and related algorithms. For convenience, we use *finding frequent/persistent/significant items* to denote finding top-k frequent/persisntent/significant items.

6.1 Metrics

Suppose the correct top-k significant set is ϕ, the reported set is ψ, consisting of $e_1, e_2, \ldots, e_k$, and the reported significance of all items is respectively $\hat{s_1}, \hat{s_2}, \ldots, \hat{s_k}$. We define the following metrics:

- ARE (average absolute error): It is defined as $\frac{1}{k} * \sum_{i=1}^{k} \frac{|s_i - \hat{s_i}|}{s_i}$, where s_i is the real significance of e_i.
- Precision: It is defined as $\frac{|\phi \cap \psi|}{|\phi|}$.
- F_1 score: Let P be $\frac{|\phi \cap \psi|}{|\phi|}$, and Q be $\frac{|\phi \cap \psi|}{|\psi|}$, the f_1 score is defined as $\frac{2*P*Q}{P+Q}$.
- Throughput: We perform insertions for all items in a dataset and divided the time by the size of the dataset.

6.2 Datasets

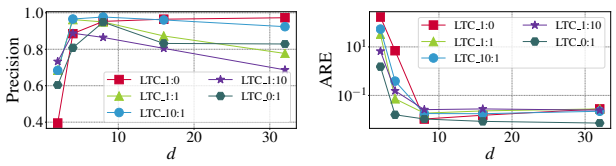
1) Social: This dataset comes from a real social network [41], which includes users' messages and the sending time. Among all the packets, we capture two consecutive days as our dataset, and we regard the username of the sender of a message as an item ID and the sending time of a message as the timestamp. This dataset contains 1.5M messages, and we divide it into 200 periods with a fixed time interval.
2) Net: This is a temporal network of interactions on the stack exchange website [42]. Each item consists of three values u, v, t, which means user u answered user v's question at time t. We regard u as an item ID and t as the timestamp. This dataset contains 10M items, and we divide it into 1000 periods with a fixed time interval.
3) CAIDA: This dataset is from *CAIDA Anonymized Internet Trace 2016* [43], consisting of IP packets. (*i.e.*, source IP address, destination IP address, source port, destination port, and protocol type). We regard the source IP address of a packet as an item ID and the index as the timestamp. This dataset contains 10M packets, and we divide it into 500 periods with a fixed time interval.

6.3 Experiment Setup

We implement our algorithm and related algorithms in C++. Here we use Bob Hash [44], [45] as our hash function. All the programs run on a server with dual 6-core CPUs (24 threads, Intel Xeon CPU E5-2620 @2 GHz) and 62GB total system memory. The source codes of LTC and other related algorithms are available on Github [46].

The compared algorithms are SS, CSS, LC, CM, CU and PIE, which are introduced in Section 2. To achieve a head-to-head comparison, we use the same memory for every algorithm except for PIE. PIE needs to maintain a Space Time Bloom Filter for each period and it cannot decode any item when the memory is tight. Therefore, we use T times of memory for PIE, where T is the number of periods, in order to make its performance comparable. For SS, CSS, LC, PIE and LTC, the number of cells is determined by the memory size. For sketch-based algorithms, we set the number of arrays to 3. When they are used to find frequent items, the size of the heap is k, and we allocate the rest memory to the sketch. When they are used to find persistent items, we need to maintain a standard BF to record whether an item has appeared in the current period, and therefore we allocate half of the memory to the standard BF and the rest memory to the sketch+heap. When they are used to find significant items, for each algorithm we maintain two sketches: one for finding frequent items, and the other for finding persistent items, and we distribute memory to them evenly. For LTC, we have conducted experiments in the next section to guide the setting of d. According to the results, we set $d = 8$ in all experiments. For every experiment, we query top-k items once at the end of insertion.

6.4 Effects of d



(a) Precision vs. d. (b) ARE vs. d.

Fig. 7: Varying d.

In this experiment, we evaluate the effects of the parameter d, where d is the number of cells in each bucket. We set k to 1000, fix the total memory to 100KB and change d. *When d increases, the performance first goes up and then goes down.* To make the experiments more comprehensive, we set five pairs of parameters: (1) $\alpha = 0, \beta = 1$, (2) $\alpha = 1, \beta = 0$, (3) $\alpha = 1, \beta = 1$, (4) $\alpha = 10, \beta = 1$, and (5) $\alpha = 1, \beta = 10$. For convenience, we use LTC_0:1, LTC_1:0, LTC_1:1, LTC_10:1, and LTC_1:10 to denote LTC on the corresponding parameters, where the first value is α and the second value is β. The dataset we use is the Net dataset. Figure 7(a) plots how precision changes when d varies from 2 to 32. Figure 7(b) plots how ARE changes when d varies from 2 to 32. We can observe that $d = 8$ always has the best performance. Therefore, we set $d = 8$ by default in the following experiments.

The reason that a small d is sub-optimal is that the capacity of a bucket is so small that collisions of significant items will be common. The reason that a large d is sub-optimal is that larger d leads to smaller w. Therefore, when an item is in the smallest cell of a bucket, it will be deducted by other items more often, which is indicated by Equation 7. In summary, there is a trade-off between w and d, and $d = 8$ is usually the optimal value.

6.5 Statistics of Significant Items in a Bucket

In this experiment, we show the statistics of LTC's buckets after processing the Net dataset. The parameter d is set to 8,

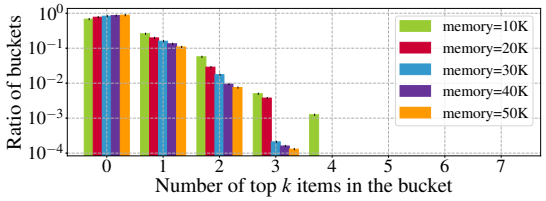
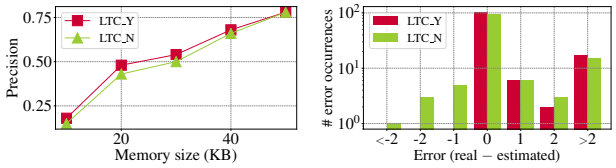


Fig. 8: Distribution of the number of significant items per bucket.

which means there are 8 cells per bucket. The top $k = 100$ items are defined as significant items. The experiment is repeated 5 times using different hash functions. As shown in Figure 8, the y-axis is the percentage of the buckets that contain x significant items, while the x-axis is x. Different columns refer to different memory sizes, ranging from 10KB to 50KB. For the Net dataset, these memory settings are very small. However, if the memory size is larger than 20K, there are less than 4 significant items colliding in the same bucket. If the memory size is larger than 30K, the possibility that a bucket contains equal or more than 3 significant items is lower than 0.1%. Since there are 8 cells in total in a bucket, we can say that hash collisions of significant items in LTC are usually slight and have a small impact on the stability of the performance.

6.6 Effects of Deviation Eliminating

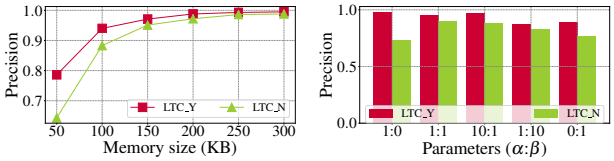


(a) Precision vs. memory size. (b) Number of different errors.

Fig. 9: Deviation eliminating (LTC_Y) vs. without (LTC_N).

In this set of experiments, we compare the version with Deviation Eliminating (LTC_Y) and the version without that (LTC_N). Here we focus on finding persistent items, which means $\alpha = 0, \beta = 1$. The dataset we use is the Net dataset. We set $k = 1000$ and vary the memory size from 10KB to 50KB. As shown in Figure 9(a), the precision of LTC_Y is slightly larger than that of LTC_N, because deviations do not often happen. The major benefit of LTC_Y is that the overestimation error of persistency is eliminated, as shown in Figure 9(b). We use this optimization by default in the following experiments.

6.7 Effects of Long-tail Restoring

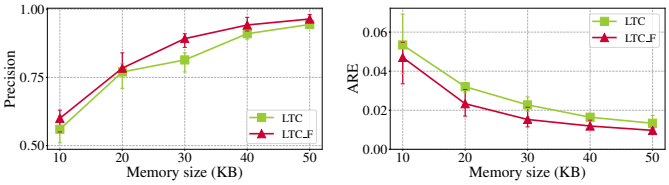


(a) Precision vs. memory size. (b) Precision vs. parameters.

Fig. 10: Long-tail Restoring (LTC_Y) vs. without (LTC_N).

In this set of experiments, we compare the LTC with Long-tail Restoring (LTC_Y) and the LTC without Long-tail Restoring (LTC_N). The dataset we use is the Net dataset. For the first experiment (Figure 10(a)), we set $\alpha = 1, \beta = 1, k = 1000$, and vary the memory size from 50KB to 300KB. For the second experiment (Figure 10(b)), we set the memory size to 50KB, $k = 1000$, and vary the parameters. The results show that the precision of LTC_Y is always larger than that of LTC_N. The reason is that the values that are lost before an item entering a cell is partially restored by Long-tail Restoring. We use Long-tail Restoring by default when we refer to LTC in the following experiments.

6.8 Effects of Frequency Based Replacement

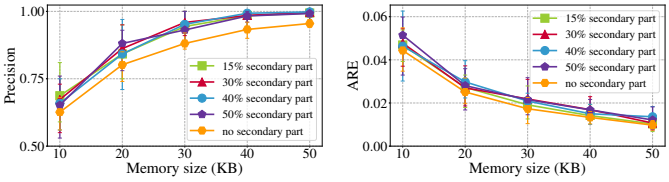


(a) Precision vs. memory size. (b) ARE vs. memory size.

Fig. 11: Frequency Based Replacement (LTC_F) vs. without (LTC).

In this set of experiments, we compare LTC with Frequency Based Replacement (LTC_F) and LTC without this optimization. The test case is finding the top k=100 significant items with parameters $\alpha = 1$, $\beta = 1$ from the Net dataset. As shown in Figure 11(a) and Figure 11(b), LTC_F is constantly better than LTC in terms of both precision and ARE when changing memory size from 10KB to 50KB. The improvement of precision is up to 10% and the improvement of ARE is up to 40%. The optimized version is better because the Frequency-only Replacement technique can protect the persistency values of the smallest items from being deducted.

6.9 Effects of Two-level Structure



(a) Precision vs. memory size. (b) ARE vs. memory size.

Fig. 12: LTCs with Two-level Structures.

In this set of experiments, we test the two-level structure with different memory divisions between the primary part and the secondary part. The tested LTCs only differ in the percentage of the secondary part. The test case is finding the top $k = 100$ significant items with parameters $\alpha = 1$,$\beta = 1$ from the Net dataset. As shown in Figure 12(a), the precision of two-level LTC is constantly better than that of one-level LTC (LTC with no secondary part), and there are up to 10% improvements. This is because many items get one more chance of not being deducted if it can enter the secondary part. However, as shown in Figure 12(b), the ARE of two-level LTCs is slightly worse than that of one-level LTC. This is because the items in the secondary part will be directly dropped if it is the smallest one in the bucket, which results in high under-estimation for a few items. Another take away is that the percentage of the secondary part does not affect the performance much, except that high percentages of secondary part have high variance as shown by the error bar in Figure 12(a).

Finally, we have to mention that the two-level structure suffers from low insertion throughput, as the measured throughput is only 50% of the one-level structure. The reason is that both two parts need to be checked for match during the insertion. As a result, we suggest to use this optimization only when high precision is preferred compared to insertion speed, and the secondary part is suggested to be set to 30% of the total memory of LTC.

6.10 Comparisons on Finding Frequent Items

In this set of experiments, we set $\alpha = 1, \beta = 0$, which means the significance of an item is only related to frequency. We compare LTC with SS, CSS, LC, CM and CU.

1) Precision vs. memory size (Figure 13(a)-(c)). *LTC has the highest precision among all data structures at all memory sizes.* In this experiment, we set $k = 100$, and vary the memory size from 5KB to 50KB. As shown in Figure 13(a), for the CAIDA dataset, when the memory size is 10KB, the precision of LC, SS, CSS, CM and CU is respectively 18%, 6%, 21%, 38% and 52%, while the one of LTC reaches 99%. As shown in Figure 13(b), for the Net dataset, when the memory size is 10KB, the precision of LC, SS, CSS, CM and CU is respectively 5%, 2%, 6%, 31% and 41%, while the one of LTC reaches 88%. As shown in Figure 13(c), for the Social dataset, when the memory size is 10KB, the precision for LC, SS, CSS, CM, CU and LTC is respectively 62%, 58%, 86%, 84%, 87% and 98%.

2) ARE vs. memory size (Figure 14(a)-(c)). *LTC has the lowest ARE among all data structures at all memory sizes.* The configuration of this experiment is the same as the previous experiment. As shown in Figure 14(a), for the CAIDA dataset, the ARE of LTC is from 2 to 6 orders of magnitude smaller than LC, SS, CSS, CM and CU. As shown in Figure 14(b), for the Net dataset, the ARE of LTC is up to 4 orders of magnitude smaller than LC, SS, CSS, CM and CU. As shown in Figure 14(c), for the Social dataset, the ARE of LTC is from 1 to 4 orders of magnitude smaller than LC, SS, CSS, CM and CU.

3) Precision vs. k (Figure 13(d)). *LTC has the highest precision among all data structures for various k.* In this experiment, we set the memory size to 100KB, and vary k from 100 to 1000. As shown in Figure 13(d), as k increases, the precision of LC, SS, CSS, CM and CU is respectively from 78% to 36%, 53% to 19%, 79% to 36%, 97% to 65% and 99% to 88%, while the one of LTC is always larger than 95%.

4) ARE vs. k (Figure 14(d). *LTC has the lowest ARE among all data structures for various k.* The configuration of this experiment is the same as the previous experiment. As shown in Figure 14(d), for the Net dataset, the ARE of LTC is respectively between 2 to 4 orders of magnitude smaller than LC, SS, CSS, CM and CU.

Analysis. Sketch-based algorithms need to record frequencies of all items, which is space-consuming, leading to poor accuracy. As for LC, SS and CSS, when the data structure is full of items, the strategy of inserting new element would lead to huge overestimation error. In contrast, LTC decrements the current smallest item before inserting a new item, and when a new item is inserted, its value is restored as accurate as possible. Therefore, LTC achieves much better performance than other related algorithms, and the above experimental results fully verify this conclusion.

6.11 Comparisons on Finding Persistent Items

In this set of experiments, we set $\alpha = 0, \beta = 1$, which means the significance of an item is only related to its persistency. We compare the performance of LTC with PIE, CM and CU.

1) Precision vs. memory size (Figure 15(a)-(c)). *LTC has the highest precision for all memory settings.* In this experiment, we set $k = 100$, and vary the memory size from 25KB to 300KB. As shown in Figure 15(a), for the CAIDA dataset, the precision of CM, CU, PIE, and LTC is respectively from 6% to 80%, 2% to 95%, 66% to 94%, and 70% to 100%. As shown in Figure 15(b), for the Net dataset, the precision of CM, CU, PIE and LTC is respectively from 4% to 86%, 2% to 99%, 64% to 99%, and 75% to 99%. As shown in Figure 15(c), for the Social dataset, all the algorithms have a high precision. Note that PIE uses T times of the memory of the other three algorithms.

2) ARE vs. memory size (Figure 16(a)-(c)). *LTC has the lowest ARE for all memory settings.* The configuration of this experiment is the same as the previous experiment. As shown in Figure 16(a), for the CAIDA dataset, the ARE of LTC is respectively between 1 to 5 orders of magnitude smaller than CM, CU and PIE. As shown in Figure 16(b), for the Net dataset, the ARE of LTC is respectively between 1 to 4 orders of magnitude smaller than CM, CU and PIE. As shown in Figure 16(c), for the Social dataset, the ARE of LTC is respectively between 6 and 25 times, 5 and 23 times smaller than CM and CU, and approximately the same as PIE.

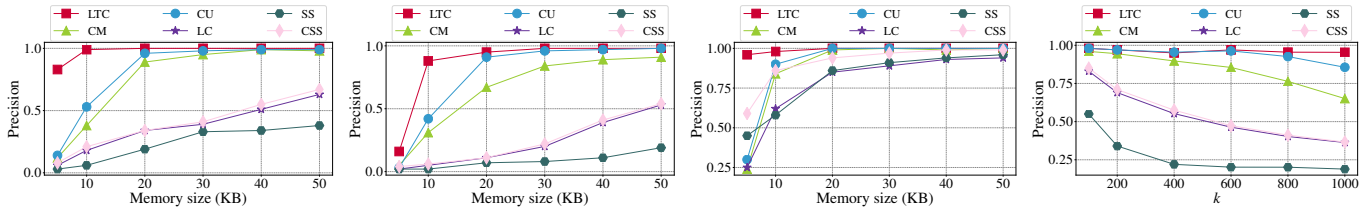
3) Precision vs. k (Figure 15(d)). *LTC has the highest precision for all k settings.* In this experiment, we set the memory size to 100KB, and vary k from 100 to 1000. As shown in Figure 15(d), when $k = 100$, the precision of CM, CU and PIE is respectively 60%, 93% and 91.5%, while the one of LTC reaches 99%. Furthermore, as k increases, the precision of LTC is always larger than 95%.

4) ARE vs. k (Figure 16(d)). *LTC has the lowest ARE for all k settings.* The configuration of this experiment is the same as the previous experiment. As shown in Figure 16(d), the ARE of LTC is respectively between 52942 and 10^8 times, 32 and 10^8 times, 7 and 50 times smaller than CM, CU and PIE.

5) Precision vs. number of periods (Figure 17(a)). *LTC has the highest precision for all settings of the number of periods.* In this experiment, we set the memory size to 100KB for LTC, CM, and CU, $k = 100$, and vary T from 500 to 3000, where T is the number of periods. As the experimental method mentioned above, when T is changed, the memory size of PIE changes but that of the other algorithms does not change. We just show the results on Net dataset. As shown in Figure 17(a), when $T = 500$, the precision of CM, CU and PIE is respectively 35%, 86% and 84%, while the one of LTC reaches 95%. Furthermore, when T is larger than 1000, the precision of LTC is always larger than 99%.

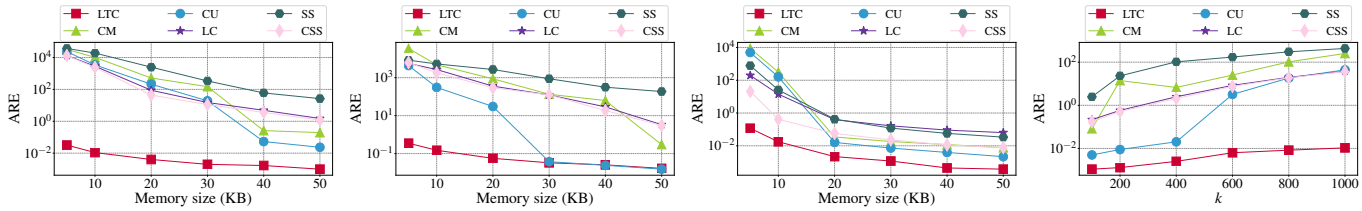
6) ARE vs. number of periods (Figure 17(b)). *LTC has the lowest ARE for all settings of the number of periods.* The configuration of this experiment is the same as the previous experiment. We just show the results on Net dataset. As shown in Figure 17(b), the ARE of LTC is respectively between 12834 and 225914 times, 12 and 87 times, 12 and 449 times smaller than CM, CU and PIE.

Analysis. Sketch-based algorithms have their shortcomings. In order to record persistencies of all items, they need to



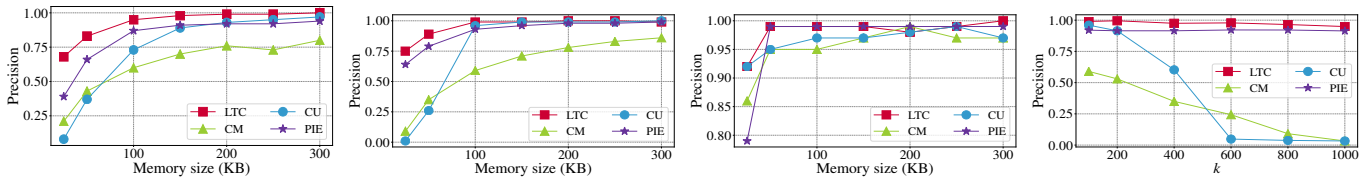
(a) Varying memory size (CAIDA). (b) Varying memory size (Net). (c) Varying memory size (Social). (d) Varying k (Net).

Fig. 13: Measuring precision on finding frequent items.



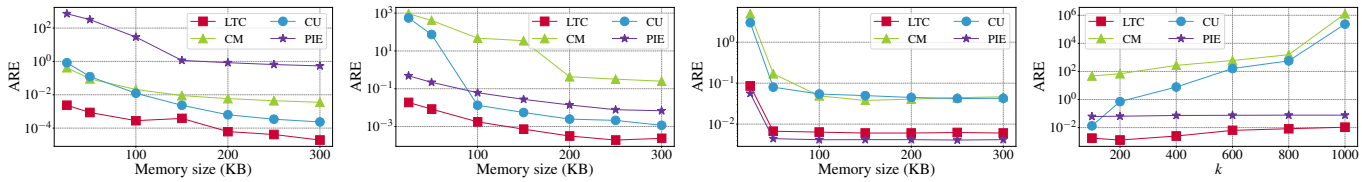
(a) Varying memory size (CAIDA). (b) Varying memory size (Net). (c) Varying memory size (Social). (d) Varying k (Net).

Fig. 14: Measuring ARE on finding frequent items.



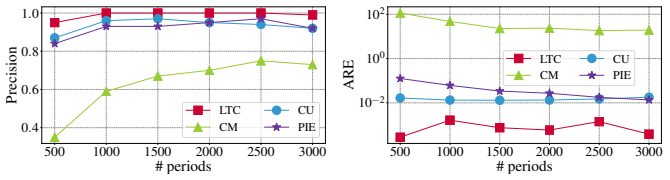
(a) Varying memory size (CAIDA). (b) Varying memory size (Net). (c) Varying memory size (Social). (d) Varying k (Net).

Fig. 15: Measuring precision on finding persistent items.



(a) Varying memory size (CAIDA). (b) Varying memory size (Net). (c) Varying memory size (Social). (d) Varying k (Net).

Fig. 16: Measuring ARE on finding persistent items.



(a) Precision (Net). (b) ARE (Net).

Fig. 17: Varying the number of periods on finding persistent items.

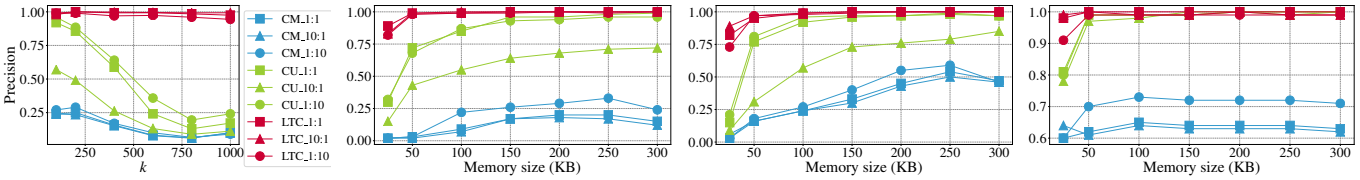
allocate about half of the size of memory to record whether an item appears in the current period. As for PIE, it needs to maintain a data structure for every period to record persistencies of all items, which also fails to achieve high memory efficiency. In contrast, LTC leverages the spirit of the well known CLOCK [31], [32] algorithm to tag period. Its overhead is only two flags (two bits) for every cell, which saves memory effectively.

6.12 Comparisons on Finding Significant Items

There is no prior work on finding significant items. Because sketch-based algorithms are the best algorithms except LTC on both finding frequent items and finding persistent items, we modify them to find significant items. To make the experiments more comprehensive, we test three pairs of weights: 1) $\alpha = 1, \beta = 10$, 2) $\alpha = 1, \beta = 1$ and 3) $\alpha = 10, \beta = 1$. We use LTC_α:β, CM_α:β and CU_α:β to denote LTC, CM and CU on the corresponding weights.

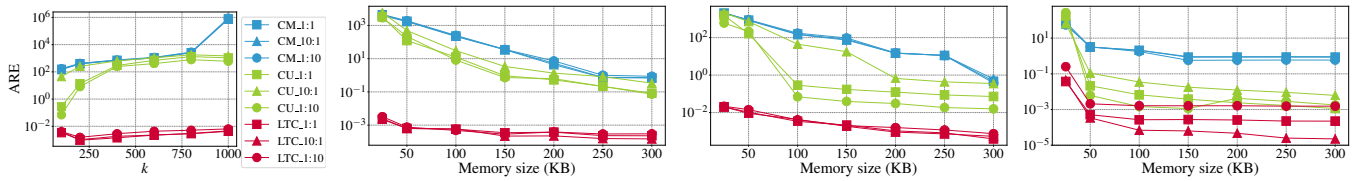
1) Precision vs. memory size (Figure 18(b)-(d)). *The precision of LTC is higher than that of CM and CU.* In this experiment, we set $k = 100$, and vary the memory size from 25KB to 300KB. As shown in Figure 18(b), for the CAIDA dataset, the precision of LTC is from 80% to 100%, while the precision of CU is from 16% to 98%. As shown in Figure 18(c), for the Net dataset, the precision of LTC is always from 70% to 100%, while the one of CU is from 10% to 99%. As shown in Figure 18(d), for the Social dataset, when the memory size is 25KB, the precision of CU is from 78% to 100%, while the one of LTC is from 90% to 100%.

2) ARE vs. memory size (Figure 19(b)-(d)). *The ARE of LTC is lower than that of CM and CU.* The configuration of this experiment is the same as the previous experiment. As shown in Figure 19(b), for the CAIDA dataset, the ARE of LTC is from 3 to 6 orders of magnitude smaller than CU. As



(a) Varying k (Net). (b) Varying memory size (CAIDA). (c) Varying memory size (Net). (d) Varying memory size (Social).

Fig. 18: Measuring precision on finding significant items.



(a) Varying k (Net). (b) Varying memory size (CAIDA). (c) Varying memory size (Net). (d) Varying memory size (Social).

Fig. 19: Measuring ARE on finding significant items.

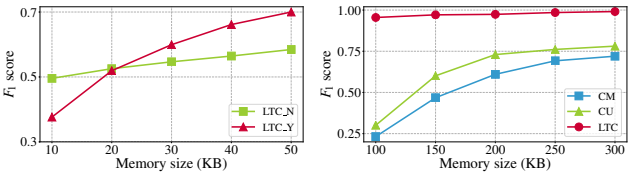
shown in Figure 19(c), for the Net dataset, the ARE of LTC is from 1 to 6 orders of magnitude smaller than CU. As shown in Figure 19(d), for the Social dataset, the ARE of LTC is up to 3 orders of magnitude smaller than CU.

3) Precision vs. k (Figure 18(a)). *The precision of LTC is higher than that of CM and CU.* In this experiment, we set the memory size to 100KB, and vary k from 100 to 1000. As shown in Figure 18(a), as k increases, the precision of CU is respectively from 93% to 23%, 53% to 17%, and 94% to 21% on each pair of parameters, while the one of LTC is always larger than 94%.

4) ARE vs. k (Figure 19(a)). *The ARE of LTC is lower than that of CM and CU.* The configuration of this experiment is the same as the previous experiment. As shown in Figure 19(a), the ARE of LTC is from 4 to 8 orders of magnitude smaller than CU on each pair of weights.

6.13 Finding Items with Significance Thresholds

The task of this experiment is defined in Section 5, which is finding items whose frequencies are at least x and persistencies are at least y. Since LTC has both false negative error and false positive error, we use f_1 score to evaluate the performance of algorithms.



(a) Effect of the Auxiliary Array. (b) LTC vs. Sketches.

Fig. 20: Finding items with significance thresholds.

1) Effects of the Auxiliary Array (Figure 20(a)). For convenience, we use LTC_Y to denote LTC with the Auxiliary Array and LTC_N to denote the LTC without the Auxiliary Array. The LTC_Y uses 20% less memory than LTC_N in order to make room for the auxiliary array.

In this set of experiments, we set x=100, y=30, and vary the memory size from 10KB to 50KB. The dataset we use is the Net Dataset. This is a stress test as there are more than 4000 items above the threshold, while there are 1000 cells in LTC when the memory size is 10K. As shown in Figure 20(a), when memory is 10KB, the auxiliary array has negative effect because it consumes the tight memory. As the memory size increases, the F_1 score of LTC_Y reaches 0.7, while F_1 of LTC_N is below 0.6. The reason that LTC_Y is better is because it can make significant items retire from LTC cells and move to the auxiliary array so that the LTC cells are always open to new items.

2) LTC vs. sketch-based algorithms (Figure 20(b)). We also conduct experiments on Net Dataset to compare LTC_Y with sketch-based algorithms. *The results show that the f_1 score of LTC is much higher than that of CM and CU.*

In this experiment, we set x=1000, y=300 to make it not too hard for sketch-based algorithms, and vary the memory size from 100KB to 300KB. As shown in Figure 20(b), as the memory size changes, the f_1 score of LTC_Y increases from 0.95 to 0.99, while the one of CM and CU increases from 0.28 to 0.73, and 0.35 to 0.79, respectively.

6.14 Experiments on Throughput

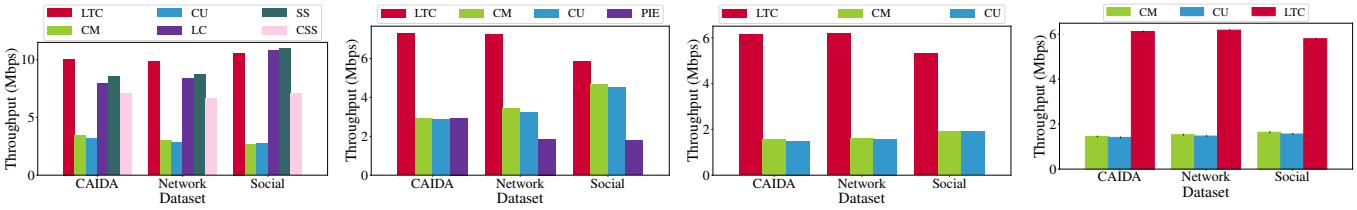
In this section, we conduct experiments to compare the throughput of all the above algorithms. *The results show that the throughput of LTC is much higher than other algorithms.*

1) Finding frequent items (Figure 21(a)). We set the memory size to 100KB and $k = 100$. As shown in Figure 21(a), the result shows that LTC is not only more accurate, but also achieves higher throughput. The reason why sketch-based algorithms are slow is that they need to maintain a min-heap whose time complexity of update is $O(log(k))$.

2) Finding persistent items (Figure 21(b)). We set the memory size to 100KB and $k = 100$. As shown in Figure 21(b), the throughput of LTC is between 5.2 and 6.8Mbps, while the one of CM, CU and PIE is respectively between 2.9 and 4.2Mbps, 2.8 and 4.1Mbps, 1.8 and 2.8Mbps.

3) Finding significant items (Figure 21(c)). We set the memory size to 100KB, $k = 100$, and $\alpha = 1, \beta = 1$. As shown in Figure 21(c), the throughput of LTC is between 5.2 and 6Mbps, while the one of CM and CU is respectively between 1.7 and 1.9Mbps, 1.6 and 1.9Mbps.

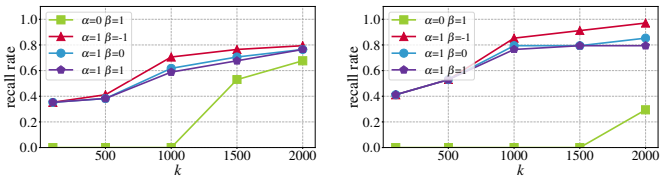
4) Finding significant items with thresholds (Figure 21(d)). We set $x = 1000$, $y = 300$, and the memory size to 100KB. As shown in Figure 21(d), the throughput of LTC is between 5.8 and 6.2Mbps, while the one of CM and CU is respectively between 1.5 and 1.7Mbps, 1.4 and 1.6Mbps.



(a) Finding frequent items. (b) Finding persistent items. (c) Finding significant items. (d) Finding significant items with thresholds.

Fig. 21: Throughput vs. dataset.

6.15 Case Study: DDoS Detection



(a) Dataset 1. (b) Dataset 2.

Fig. 22: DDoS detection with LTC.

In this case study, we show how finding significant items can help address a real world issue, DDoS detection. The network flows are originally collected by Canadian Institute for Cybersecurity [47] and are then organized as two datasets [48], denoted by Dataset 1 and Dataset 2. Dataset 1 has 8M flows from 25K IP addresses, while Dataset 2 has 13M flows from 36K IP addresses. The recall rate is the number of DDoS attacker IP addresses in the top-k significant items divided by the number of all DDoS attacker IP addresses, Experiments are conducted on LTCs with different α and β parameters. As shown in Figure 22(a) and Figure 22(b), LTC with α=1 and β=-1 (high frequency, low persistency) has the highest recall rate on both Dataset 1 and Dataset 2, followed by LTC with α=1 and β=0 (high frequency only) and LTC with α=1 and β=1 (high frequency and high persistency). LTC with α=0 and β=1 (high persistency only) has the worst performance.

The logic behind the results is that DDoS attackers usually send requests in a burst. Therefore, although benign IP addresses and DDoS IP addresses are both frequent, they can be differentiated by the persistency. If an IP address is very frequent but not persistent, it has a high probability to be a DDoS attacker. This case study also reveals the flexibility of LTC: it can support not only positive weights but also negative weights. For other applications, we suggest exploring the optimal weights guided by domain knowledge.

7 Conclusion

In this paper, we abstracted a problem named finding top-k significant items, encountered in many applications but not studied before. To accurately find top-k significant items with small memory size, we proposed a new algorithm, namely LTC. It has two key techniques, Long-tail Restoring and an adapted CLOCK algorithm, as well as three optimizations including extension of significance, frequency based replacement and two-level structure In addition, we adapted LTC to find significant items with thresholds by using an auxiliary array. We derived the theoretical bound of the correct rate and the error rate of LTC. Also, extensive experiments on 3 real datasets showed that LTC achieved high accuracy and high speed with small memory, outperforming all related algorithms. Moreover, a case study on DDoS detection showed that items' significance captured more information than items' frequency and would gain higher detection rate. The source codes of LTC and other related algorithms are available on GitHub [46].

Acknowledgement

This work is supported by National Key RD Program of China (No. 2018YFB1004403), National Natural Science Foundation of China (NSFC) (No. 61832001, 61702016, 61672061), PKU-Baidu Fund 2019BD006 and Beijing Academy of Artificial Intelligence (BAAI), and the project of "FANet: PCL Future Greater-Bay Area Network Facilities for Large-scale Experiments and Application" (No. LZC0019).

References

[1] Tongya Zheng, Gang Chen, Xinyu Wang, Chun Chen, Xingen Wang, and Sihui Luo. Real-time intelligent big data processing: technology, platform, and applications. *Science China Information Sciences*, 62(8):82101, 2019.

[2] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7), 1970.

[3] Saar Cohen and Yossi Matias. Spectral bloom filters. In *ACM SIGMOD International Conference on Management of Data*, 2003.

[4] Tong Yang, Alex X. Liu, Muhammad Shahzad, Yuankun Zhong, Qiaobin Fu, Zi Li, Gaogang Xie, and Xiaoming Li. A shifting bloom filter framework for set queries. *Vldb*, 9(5), 2016.

[5] Michael Mitzenmacher. Compressed bloom filters. *IEEE/ACM transactions on networking*, 10(5), 2002.

[6] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. *Automata, languages and programming*, 2002.

[7] Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1), 2005.

[8] Cristian Estan and George Varghese. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. *ACM Transactions on Computer Systems (TOCS)*, 21(3), 2003.

[9] Pratanu Roy, Arijit Khan, and Gustavo Alonso. Augmented sketch: Faster and more accurate stream processing. In *Proc. SIGMOD*, 2016.

[10] Tong Yang, Yang Zhou, Hao Jin, Shigang Chen, and Xiaoming Li. Pyramid sketch: a sketch framework for frequency estimation of data streams. *Proceedings of the VLDB Endowment*, 10(11), 2017.

[11] Jizhou Luo, Wei Zhang, Shengfei Shi, Hong Gao, Jianzhong Li, Wei Wu, and Shouxu Jiang. Freshjoin: An efficient and adaptive algorithm for set containment join. *Data Science and Engineering*, 4(4):293–308, 2019.

[12] Katsiaryna Mirylenka, Graham Cormode, Themis Palpanas, and Divesh Srivastava. Conditional heavy hitters: detecting interesting correlations in data streams. *The VLDB Journal*, 24(3), 2015.

[13] Joong Hyuk Chang and Won Suk Lee. Finding recent frequent itemsets adaptively over online data streams. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2003.

[14] Yin-Ling Cheung and Ada Wai-Chee Fu. Mining frequent itemsets without support threshold: with and without item constraints. *IEEE Transactions on Knowledge and Data Engineering*, 16(9), 2004.

[15] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *Proc. Springer ICDT*, 2005.

[16] Ran Ben-Basat, Gil Einziger, Roy Friedman, and Yaron Kassner. Heavy hitters in streams and sliding windows. In *Proc. IEEE INFOCOM*, 2016.

[17] Gurmeet Singh Manku and Rajeev Motwani. Approximate frequency counts over data streams. In *Proc. VLDB*, 2002.

[18] Haipeng Dai, Muhammad Shahzad, Alex X. Liu, and Yuankun Zhong. Finding persistent items in data streams. *Proceedings of the Vldb Endowment*, 10(4):289–300, 2016.

[19] Sneha Aman Singh and Srikanta Tirthapura. Monitoring persistent items in the union of distributed streams. *Journal of Parallel and Distributed Computing*, 74(11):3115–3127, 2014.

[20] Jelena Mirkovic and Peter Reiher. A taxonomy of ddos attack and ddos defense mechanisms. *Acm Sigcomm Computer Communication Review*, 34(2):39–53, 2004.

[21] Ruifeng Xu, Jiachen Du, Zhishan Zhao, Yulan He, Qinghong Gao, and Lin Gui. Inferring user profiles in social media by joint modeling of text and networks. *Science China Information Sciences*, 62(11):219104, 2019.

[22] Stephen Bonner, Ibad Kureshi, and et al. Exploring the semantic content of unsupervised graph embeddings: An empirical study. *Data Science and Engineering*, 4(3):269–289, 2019.

[23] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. The nature of data center traffic: measurements & analysis. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*. ACM, 2009.

[24] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. Inside the social network's (datacenter) network. In *ACM SIGCOMM Computer Communication Review*. ACM, 2015.

[25] Mark Allman, Vern Paxson, and Ethan Blanton. Tcp congestion control. Technical report, 2009.

[26] Anja Feldmann and Ward Whitt. Fitting mixtures of exponentials to long-tail distributions to analyze network performance models . *Performance Evaluation*, 31(34):1096–1104 vol.3, 1997.

[27] Joseph Abate, Gagan L. Choudhury, and Ward Whitt. Waiting-time tail probabilities in queues with long-tail service-time distributions. *Queueing Systems*, 16(3-4):311–338, 1994.

[28] Gagan L. Choudhury and Ward Whitt. Long-tail buffer-content distributions in broadband networks. *Performance Evaluation*, 30(3):177–190, 1997.

[29] Allen B. Downey. Evidence for long-tailed distributions in the internet. In *ACM SIGCOMM Internet Measurement Workshop*, pages 229–241, 2001.

[30] N. G. Duffield and W. Whitt. Network design and control using on-off and multi-level source traffic models with long-tailed distributions. *At & T Labs*, pages 421–445, 1997.

[31] Gunner Danneels. System for synchronizing data stream transferred from server to client by initializing clock when first packet is received and comparing packet time information with clock, 1997.

[32] Nicholas A. J Millington. System and method for synchronizing operations among a plurality of independently clocked digital data processing devices, 2015.

[33] Bin Fan, David G Andersen, and Michael Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *NSDI*, volume 13, 2013.

[34] Haipeng Dai, L Meng, and Alex X Liu. Finding persistent items in distributed, datasets. In *Proc. IEEE INFOCOM*, 2018.

[35] Qingchun Meng and Xiaojing Wang. Research on precoding method in raptor code. *Computer Engineering*, 33(1):1–3, 2007.

[36] Haipeng Dai, Yuankun Zhong, Alex X Liu, Wei Wang, and Meng Li. Noisy bloom filters for multi-set membership testing. In *Proc. ACM SIGMETRICS*, pages 139–151, 2016.

[37] Alan Jay Smith. Sequentiality and prefetching in database systems. *ACM Transactions on Database Systems (TODS)*, 3(3):223–247, 1978.

[38] Zhetao Li, Baoming Chang, Shiguo Wang, Anfeng Liu, Fanzi Zeng, and Guangming Luo. Dynamic compressive wide-band spectrum sensing based on channel energy reconstruction in cognitive internet of things. *IEEE Transactions on Industrial Informatics*, 2018.

[39] Zhetao Li, Fu Xiao, Shiguo Wang, Tingrui Pei, and Jie Li. Achievable rate maximization for cognitive hybrid satellite-terrestrial networks with af-relays. *IEEE Journal on Selected Areas in Communications*, 36(2):304–313, 2018.

[40] Zhetao Li, Yuxin Liu, Anfeng Liu, Shiguo Wang, and Haolin Liu. Minimizing convergecast time and energy consumption in green internet of things. *IEEE Transactions on Emerging Topics in Computing*, 2018.

[41] The Social dataset Traces. http://open.weibo.com/wiki/2/friendships/friends/.

[42] The StackExchange dataset Traces. http://snap.stanford.edu/data/.

[43] The CAIDA Anonymized Internet Traces. http://www.caida.org/data/overview/.

[44] The source code of Bob Hash. http://burtleburtle.net/bob/hash/evahash.html.

[45] Christian Henke, Carsten Schmoll, and Tanja Zseby. Empirical evaluation of hash functions for multipoint measurements. *SIGCOMM Comput. Commun. Rev.*, 38(3), July 2008.

[46] The source code of ltc and related algorithms. https://github.com/shiyu-cheng/Finding-Significant-Items.

[47] Iman Sharafaldin, Arash Habibi Lashkari, and Ali A Ghorbani. Toward generating a new intrusion detection dataset and intrusion traffic characterization. In *ICISSP*, pages 108–116, 2018.

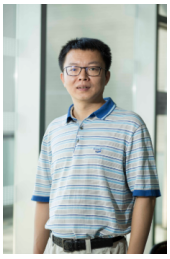
[48] The parsed DDoS Dataset. https://www.kaggle.com/devendra416/ddos-datasets.

Shiyu Cheng is a master student in the Faculty of Geographical Science at Beijing Normal University. She is a research intern in the Network Big Data Lab at Peking University under the guidance of professor Tong Yang. She is working on the intersection of computer science and remote sensing. Her research interest is big data analysis and green infrastructure in cities.

Dongsheng Yang is currently a Ph.D. student in the Computer Science Department at Princeton University. He got his Master's degree in Computer Science from Carnegie Mellon University in 2019 and his Bachelor's degree in Computer Science and Technology from Peking University in 2018. Dongsheng is interested in machine learning, data intensive systems, and data stream algorithms.

Tong Yang received his PhD degree in Computer Science from Tsinghua University in 2013. He visited Institute of Computing Technology, Chinese Academy of Sciences (CAS). Now he is a research assistant professor in Computer Science Department, Peking University. His research interests include network measurements, sketches, IP lookups, Bloom filters, sketches and KV stores. He published papers in SIGCOMM, SIGKDD, SIGMOD, SIGCOMM CCR, VLDB, ATC, ToN, ICDE, INFOCOM, *etc.*

Haowei Zhang is an undergraduate at Peking University. His mentor is Tong Yang. He is interested in the field of filters and sketches.



Bin Cui is a professor in the School of EECS and director of Institute of Network Computing and Information Systems, Peking University. His research interests include database system architectures, query and index techniques, big data management, and mining. He is currently on the editorial boards of the VLDB Journal, Distributed and Parallel Databases Journal, Information Systems, and Frontier of Computer Science. He is the winner of the Microsoft Young Professorship award and the CCF Young Scientist award.